INTER-PROCESS COMMUNICATION AND SYNCHRONISATION:

Lesson-11: Priority Inversion Problem and Deadlock Situations

1. Priority Inversion

Assume

- Priorities of tasks be in an order such that task *I* highest priority, task *J* a lower, and task *K* the lowest priority.
- Only tasks *I* and *K* share the data and *J* does not share data with K.
- Also let tasks *I* and *K* alone share a semaphore s_{ik} and not *J*.

Few tasks share a semaphore

- Why do only a few tasks share a semaphore? Can't all share a semaphore?
- Answer─ Worst-case latency becomes too high and may exceed the deadline if all tasks are blocked when one task takes a semaphore.
- The worst-case latency will be small only if the time taken by the tasks that share the resources is relevant

Priority Inversion Situation

- At an instant t0, suppose task *K* takes s_{ik} , it does not block task *J* and blocks only the task *I*.
- This happens because only tasks *I* and *K* share the data and *J* does not and I is blocked at instance t0 due to wait for some message and *sik* .
- Consider the problem that now arises on selective sharing between *K* and *I*.

Priority Inversion Situation

- At next instant t1, let task *K* become ready first on an interrupt.
- Now, assume that at next instant t2, task *I* becomes ready on an interrupt or getting the waiting message.
- At this instant, *K* is in the critical section.
- Therefore, task *I* cannot start at this instant due to K being in the critical region.

Priority Inversion Situation …

- Now, if at next instant t3, some action (event) causes the unblocked higher than K priority task *J* to run.
- After instant t3, running task *J* does not allow the highest priority task *I* to run because *K* is not running, and therefore *K* can't release s_{ik} that it shares with *I*.

Priority Inversion Situation …

- Further, the design of task *J* may be such that even when s_{ik} is released by task *K*, it may not let *I* run. [*J* runs the codes as if it is in critical section all the time after executing DI.] The *J* action is now as if *J* has higher priority than *I*. This is because K, after entering the critical section and taking the semaphore the OS let the *J* run, did not share the priority information about *I*—that task *I* is of higher priority than *J*.

Priority Inversion Situation …

- The priority information of another higher-priority task *I* should have also been inherited by *K* temporarily, if *K* waits for *I* but *J* does not and *J* runs when *K* has still not finished the critical section codes.
- This did not happen because the given OS was such that it didn't provide for temporary priority inheritance in such situations.
- Above situation is also called a ***priority inversion problem***

OS Provision for temporary priority inheritance in such situations

- Some OSes provide for priority inheritance in these situations and thus priority inheritance problem does not occur when using them.
- A mutex should be a mutually exclusive Boolean function, by using which the critical section is protected from interruption in such a way that the problem of priority inversion does not arise.

OS Provision for temporary priority inheritance in such situations

- Mutex is provided in certain RTOS so that it the priority inversion problem does not arise.
- If OS mutex functions, not provided then programmer uses the available semaphore functions, such that initailised value = 1 and at the start of the critical section, pend (to make it 0 (unavailable) and at the end post function (available) is used

OS Provision for temporary priority inheritance in such situations

- Mutex is automatically provided with priority inheritance by task on taking it in certain OSes so that it the priority inversion problem does not arise and certain OSes provides for selecting priority inheritance as well as priority sealing options.

2. Deadlock Situation

Assume

- Priorities of tasks be such that task *H* is of highest priority, task *I* a lower priority and task *J* the lowest.
- Two semaphores, *SemTok1* and *SemTok2*.
- Tasks *I* and *H* have a shared resource through *SemTok1* only.
- Tasks *I* and *J* have two shared resources through two semaphores, *SemTok1* and *SemTok2*.

Deadlock Situation

- At a next instant t1, being now of a higher priority, the task *H* interrupts the task *I* and *J* after it takes the semaphore *SemTok1*, and thus blocks both *I* and *J*.

Deadlock Situation

- In between the time interval t0 and t1, the *SemTok1* was released but *SemTok2* was not released during the run of task *J*. But the latter did not matter as the tasks *I* and *J* don't share *SemTok2*.

Deadlock Situation

- At an instant t2, if *H* now releases the *SemTok1*, lets the task *I* take it.
- Even then it cannot run because it is also waiting for task *J* to release the *SemTok2*.
- The task *J* is waiting at a next instant t3, for either *H* or *I* to release the *SemTok1 because it needs this to again enter a critical section in it.*
- Neither task *I* can run after instant t3 nor task *J*.

Deadlock Situation Solution

- There is a circular dependency established between *I* and *J*.

Deadlock Situation solution

- On the interrupt by *H*, the task *J*, before exiting from the running state, should have been put in queue-front so that later on, it should first take *SemTok1*, and the task *I* put in queue next for the same token, then the deadlock would not have occurred

Summary

We learnt

- Priority becomes inverted and deadlock (circular dependency) develops in certain situations when using semaphores.
- Certain OSes provide the solution to this problem of semaphore use by ensuring that these situations do not arise during the concurrent processing of multitasking operations

We learnt

- Deadlock means occurrence of circular dependency

End of Lesson11 of Chapter 9 on Priority Inversion Problem and Deadlock Situations